Input Validation in PHP

Securing Your Applications with Validator.php

Validation

- Using a form and the POST method, users send their information.
- In this example, the users are required to input the name with 2 - 50 characters.

```
<form method="post">
    <h3>User Registration</h3>
    <div>
        <label>Name (2–50 chars):</label><br>
        <input type="text"
               name="name"
               value="<?= htmlspecialchars($_POST['name'] ?? '') ?>"
        >
    </div>
</form>
```

- The PHP server should check if the user's name follows the rule.
- This example shows the idea of validation.

```php
$leng = strlen($_POST['name']);
if ($leng >= 2 && $leng <= 50>) {
    // PROCESS
}
else {
    // ERROR
}
```

Why Validation Matters

Without Validation

```php
// ❌ Dangerous - no validation
$email = $_POST['email'];
$sql = "INSERT INTO users (email) VALUES ('$email')";
```

What could go wrong?

- **SQL Injection**: `'; DROP TABLE users; --`
- **XSS Attacks**: `<script>alert('hacked')</script>`
- **Data Corruption**: Invalid emails, negative ages
- **Security Breaches**: Malicious file uploads

With Validation

```
// ✅ Safe - proper validation
$validator = new Validator();
$validator->required($_POST['email'], 'Email')
          ->email($_POST['email'], 'Email');

if (!$validator->hasErrors()) {
    // Safe to use data
}
```

Types of Validation

1. Required Fields

```php
$name = $_POST['name'] ?? '';
if (empty($name)) {
    $errors[] = "Name is required";
}
```

2. Format Validation

```php
$email = $_POST['email'] ?? '';
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    $errors[] = "Invalid email format";
}
```

3. Length Validation

```php
$password = $_POST['password'] ?? '';
if (strlen($password) < 6) {
    $errors[] = "Password must be at least 6 characters";
}
```

4. Range Validation

```php
$age = $_POST['age'] ?? 0;
if ($age < 18 || $age > 120) {
    $errors[] = "Age must be between 18 and 120";
}
```

Simple Validation Example

Without Validator Class

```php
<?php
$errors = [];
if ($_POST) {
    $name = $_POST['name'] ?? '';
    $email = $_POST['email'] ?? '';
    $age = $_POST['age'] ?? '';
    // Check required fields
    if (empty($name)) { $errors[] = "Name is required"; }
    if (empty($email)) { $errors[] = "Email is required"; }
    // Check email format
    if (!empty($email) && !filter_var($email, FILTER_VALIDATE_EMAIL)) {
        $errors[] = "Invalid email format";
    }
    ...
}
?>
```

```
<form method="post">
    Name: <input type="text" name="name" value="<?= htmlspecialchars($_POST['name'] ?? '') ?>"><br>
    Email: <input type="email" name="email" value="<?= htmlspecialchars($_POST['email'] ?? '') ?>"><br>
    Age: <input type="number" name="age" value="<?= htmlspecialchars($_POST['age'] ?? '') ?>"><br>
    <button type="submit">Register</button>
</form>

<?php if (!empty($errors)): ?>
    <div style="color: red;">
        <?php foreach ($errors as $error): ?>
            <div><?= htmlspecialchars($error) ?></div>
        <?php endforeach; ?>
    </div>
<?php endif; ?>
```

- PHP feature in this example
  - The `<?= ... ?>` syntax in PHP is a shorthand for `<?php echo ... ?>`.
  - It automatically outputs the value of the contained expression.

Issues with this approach

- If you don’t use a Validator class and instead write procedural, inline validation logic like in your example, you run into several issues:
- Why This Approach is a Bad Idea
    i. Code Duplication
    ii. Poor Maintainability
    iii. No Reusability
    iv. Hard to Extend
    v. Error Handling is Inconsistent
    vi. Difficult to Test
    vii. Violates Single Responsibility Principle (SRP)

- Introducing a Validator class makes your code cleaner, reusable, testable, and easier to maintain.
- Without it, you risk building a system that’s brittle, repetitive, and prone to inconsistent validation behavior.

Validator class

Basic Structure

A Validator class is typically structured to encapsulate all validation logic into one reusable component. It usually has:

1. Properties

- To hold the data being validated (e.g., form input).
- To store error messages encountered during validation.

2. Public Methods for Common Validation Rules

- Each method checks a specific rule and, if the rule fails, adds an error message to the error list.

```php
class Validator {
    private $errors = [];
    public function required($value, $field_name) {
        if (empty($value)) {
            $this->errors[] = "$field_name is required";
        }
        return $this; // For method chaining
    }
    public function email($value, $field_name) {
        if (!empty($value) && !filter_var($value, FILTER_VALIDATE_EMAIL)) {
            $this->errors[] = "$field_name must be a valid email";
        }
        return $this;
    }
    public function hasErrors() { return !empty($this->errors); }
    public function getErrors() { return $this->errors; }
```

Error Handling Methods

```php
class Validator {
    // ... validation methods ...

    public function hasErrors() {return !empty($this->errors); }

    public function getErrors() { return $this->errors; }

    public function getFirstError() { return $this->errors[0] ?? null; }

    public function getErrorsAsString($separator = '<br>') { return implode($separator, $this->errors); }

    public function clear() {
        $this->errors = [];
        return $this;
    }

    public function count() { return count($this->errors); }
}
```

```php
// Usage examples:
if ($validator->hasErrors()) {
    echo "Found " . $validator->count() . " errors:<br>";
    echo $validator->getErrorsAsString();

    // Or just show first error
    echo "Error: " . $validator->getFirstError();
}
```

Utility functions

```
    public function minLength($value, $min, $field_name) {
        if (!empty($value) && strlen($value) < $min) {
            $this->errors[] = "$field_name must be at least $min characters";
        }
        return $this;
    }

    public function maxLength($value, $max, $field_name) {
        if (!empty($value) && strlen($value) > $max) {
            $this->errors[] = "$field_name must be no more than $max characters";
        }
        return $this;
    }

    public function numeric($value, $field_name) {
        if (!empty($value) && !is_numeric($value)) {
            $this->errors[] = "$field_name must be numeric";
        }
        return $this;
    }
```

Method Chaining

```php
$validator = new Validator();

// ✅ Chain multiple validations
$validator->required($_POST['name'], 'Name')
          ->minLength($_POST['name'], 2, 'Name')
          ->maxLength($_POST['name'], 50, 'Name');

$validator->required($_POST['email'], 'Email')
          ->email($_POST['email'], 'Email');
```

```php
$validator->required($_POST['password'], 'Password')
          ->minLength($_POST['password'], 6, 'Password');

// Check all validations at once
if ($validator->hasErrors()) {
    foreach ($validator->getErrors() as $error) {
        echo $error . "<br>";
    }
}
```

- We can combine multiple validations in sequence to efficiently perform comprehensive checks.

Why Method Chaining?

- **Readable**: Easy to understand validation rules
- **Flexible**: Add or remove validations easily
- **Efficient**: One validator instance for all fields

Pattern Matching

```php
public function pattern($value, $pattern, $field_name, $error_message = null) {
    if (!empty($value) && !preg_match($pattern, $value)) {
        $message = $error_message ?? "$field_name format is invalid";
        $this->errors[] = $message;
    }
    return $this;
}

// Usage:
$validator->pattern($_POST['phone'], '/^\d{3}-\d{3}-\d{4}$/', 'Phone',
                    'Phone must be in format: 123-456-7890');
```

Validation of Properties

Username Validation

```
public function username($value, $field_name = 'Username') {
    if (!empty($value) && !preg_match('/^[a-zA-Z0-9_-]{3,20}$/', $value)) {
        $this->errors[] = "$field_name must be 3-20 characters, letters, numbers, underscore, or hyphen only";
    }
    return $this;
}

// Usage:
$validator->username($_POST['username']);
```

Numeric Range

```
public function min($value, $min, $field_name) {
    if (!empty($value) && is_numeric($value) && $value < $min) {
        $this->errors[] = "$field_name must be at least $min";
    }
    return $this;
}

public function max($value, $max, $field_name) {
    if (!empty($value) && is_numeric($value) && $value > $max) {
        $this->errors[] = "$field_name must be no more than $max";
    }
    return $this;
}
```

Validator Examples

basic_example.php

- Checks users' inputs: name, email, and age using the validator.

```php
<?php
require_once 'Validator.php';

$validator = new Validator();

if ($_POST) {
    // Method chaining - validate multiple rules for each field
    $validator->required($_POST['name'] ?? '', 'Name')
              ->minLength($_POST['name'] ?? '', 2, 'Name')
              ->maxLength($_POST['name'] ?? '', 50, 'Name');
    $validator->required($_POST['email'] ?? '', 'Email')
              ->email($_POST['email'] ?? '', 'Email');
    $validator->required($_POST['age'] ?? '', 'Age')
              ->numeric($_POST['age'] ?? '', 'Age')
              ->min($_POST['age'] ?? 0, 18, 'Age')
              ->max($_POST['age'] ?? 0, 120, 'Age');
```

- Process if valid

```
    if (!$validator->hasErrors()) {
        echo "<div style='color: green; padding: 10px; background: #eeffee; margin: 10px 0;'>
                <strong>Registration successful using Validator class!</strong><br>
                Name: " . htmlspecialchars($_POST['name']) . "<br>
                Email: " . htmlspecialchars($_POST['email']) . "<br>
                Age: " . htmlspecialchars($_POST['age']) . "<br>
                <em>Found " . $validator->count() . " errors (should be 0)</em>
              </div>";
    }
}
?>
```

registration_form.php

- The validator checks users' input when users send the information via a form to register.

```php
<?php
require_once 'Validator.php';
$validator = new Validator();
if ($_POST) {
    // Validate all fields with method chaining
    $validator->required($_POST['username'] ?? '', 'Username')->username($_POST['username'] ?? '');
    ...
    // Optional field validation
    if (!empty($_POST['website'])) {
        $validator->pattern($_POST['website'], '/^https?:\/\/.+/', 'Website',
                            'Website must start with http:// or https://');
    }
    // Process if valid
    if (!$validator->hasErrors()) {
        echo "<div style='color: green;'>Registration successful!</div>";
        // Save to database...
    }
}
?>
```

- When errors occur, we display them on the screen.

```
<!DOCTYPE html>
<html>
<head>
    <title>Registration Form</title>
</head>
<body>
    <h2>User Registration</h2>

    <?php if ($validator->hasErrors()): ?>
        <div class="error">
            <strong>Please fix the following errors:</strong><br>
            <?php foreach ($validator->getErrors() as $error): ?>
                <div><?= htmlspecialchars($error) ?></div>
            <?php endforeach; ?>
        </div>
    <?php endif; ?>
```

```
    <form method="post">
        <div class="form-group">
            <label>Username:</label><br>
            <input type="text" name="username" value="<?= htmlspecialchars($_POST['username'] ?? '') ?>" required>
        </div>

        <div class="form-group">
            <label>Email:</label><br>
            <input type="email" name="email" value="<?= htmlspecialchars($_POST['email'] ?? '') ?>" required>
        </div>

        <div class="form-group">
            <label>Password:</label><br>
            <input type="password" name="password" required>
        </div>

        <div class="form-group">
            <label>Age:</label><br>
            <input type="number" name="age" value="<?= htmlspecialchars($_POST['age'] ?? '') ?>" required>
        </div>

        <div class="form-group">
            <label>Website (optional):</label><br>
            <input type="url" name="website" value="<?= htmlspecialchars($_POST['website'] ?? '') ?>">
        </div>

        <button type="submit">Register</button>
    </form>
</body>
</html>
```

api_validation.php

- Users make a POST request.
- Check the users' input using the validator.

```php
<?php
require_once 'Validator.php';

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    header('Content-Type: application/json');
$input = json_decode(file_get_contents('php://input'), true);

    $validator = new Validator();

    // Validate API input
    $validator->required($input['name'] ?? '', 'Name')
              ->minLength($input['name'] ?? '', 2, 'Name')
              ->maxLength($input['name'] ?? '', 100, 'Name');
    ...
```

- Return error or valid data depending on the validation results.

```
    // Return validation errors
    if ($validator->hasErrors()) {
        http_response_code(400);
        echo json_encode([
            'success' => false,
            'errors' => $validator->getErrors(),
            'message' => 'Validation failed'
        ]);
        exit;
    }

    // Process valid data
    echo json_encode([
        'success' => true,
        'message' => 'User created successfully',
        'data' => [
            'name' => $input['name'],
            'email' => $input['email'],
            'age' => (int)$input['age']
        ]
    ]);
}
?>
```

file_upload.php

- FileValidator class extends the existing Validator to support special file-related functions.
- File uploading requires a special process.
- It is essential to understand how files are uploaded using server-side PHP code and client-side HTML code to make a correct validation file upload algorithm.

HTML client-side file upload

- The input type "file" allows users to choose a file and upload it to the PHP server.
- The name="image" specifies where to find the uploaded image.

```
<div class="form-container">
    <form method="post" enctype="multipart/form-data">
        <div class="form-group">
            ...
            <input type="file"
                   name="image" accept="image/*" required>
            ...
        </div>
        </div>
```

PHP server-side file upload

- From the method (POST), and name ("image"), the PHP server can process the uploaded image in $_FILES['image'].

```
if ($_POST) {
    // Validate file upload
    if (isset($_FILES['image'])) {
        $_FILES['image'], ...
    } else {
        ...
    }
```

- Make a directory to upload the file to the server.
- Using the PHP move_uploaded_file() function, we move the file to the directory.

```php
$upload_dir = 'uploads/';
if (!is_dir($upload_dir)) {
    mkdir($upload_dir, 0755, true);
}

$filename = uniqid() . '_' . $_FILES['image']['name'];
$filepath = $upload_dir . $filename;

// tmp_name is automatically generated
if (move_uploaded_file($_FILES['image']['tmp_name'], $filepath)) {
```

FileValidator class

- Step 1: Checking file existence
- Step 2: Check file size

```php
class FileValidator extends Validator {
    // ... existing methods ...

    // Check file existence
    public function fileUpload($file, $field_name, $options = []) {
        if (!isset($file) || $file['error'] !== UPLOAD_ERR_OK) {
            $this->errors[] = "$field_name upload failed";
            return $this;
        }

        // Check file size
        $max_size = $options['max_size'] ?? 5242880; // 5MB default
        if ($file['size'] > $max_size) {
            $this->errors[] = "$field_name must be smaller than " . ($max_size / 1024 / 1024) . "MB";
        }
```

- Step 3: Check file size
- Step 4: Check file extension

```
        // Check file type
        $allowed_types = $options['allowed_types'] ?? ['image/jpeg', 'image/png', 'image/gif'];
        if (!in_array($file['type'], $allowed_types)) {
            $this->errors[] = "$field_name must be a valid image file";
        }

        // Check file extension
        $file_ext = strtolower(pathinfo($file['name'], PATHINFO_EXTENSION));
        $allowed_exts = $options['allowed_extensions'] ?? ['jpg', 'jpeg', 'png', 'gif'];
        if (!in_array($file_ext, $allowed_exts)) {
            $this->errors[] = "$field_name must have a valid extension";
        }

        return $this;
    }
}
```

Usage example

1. Validate form fields

```
if ($_POST) {
    // Validate regular form fields
    $validator->required($_POST['title'] ?? '', 'Title')
              ->minLength($_POST['title'] ?? '', 3, 'Title')
              ->maxLength($_POST['title'] ?? '', 100, 'Title');

    $validator->required($_POST['description'] ?? '', 'Description')
              ->maxLength($_POST['description'] ?? '', 500, 'Description');
```

2. Validate file upload

```
    if (isset($_FILES['image'])) {
        $validator->fileUpload($_FILES['image'], 'Image', [
            'max_size' => 2097152, // 2MB
            'allowed_types' => ['image/jpeg', 'image/png', 'image/gif'],
            'allowed_extensions' => ['jpg', 'jpeg', 'png', 'gif']
        ]);
    } else {
        $validator->errors[] = "Image file is required";
    }
```

3. Process if valid

```
    if (!$validator->hasErrors()) {
        // In a real app, you would move the uploaded file and save it to the database
        $upload_dir = 'uploads/';
        if (!is_dir($upload_dir)) {
            mkdir($upload_dir, 0755, true);
        }

        $filename = uniqid() . '_' . $_FILES['image']['name'];
        $filepath = $upload_dir . $filename;

        if (move_uploaded_file($_FILES['image']['tmp_name'], $filepath)) {
            $upload_success = true;
            $uploaded_file = $filepath;
        } else {
            $validator->errors[] = "Failed to save uploaded file";
        }
    }
}
```

Custom_validation.php

- We can extend the Valditor to support any custom validator.

```
class CustomValidator extends Validator {
    // ... existing methods ...

    public function unique($value, $table, $column, $field_name, $exclude_id = null) {
        if (empty($value)) return $this;

        // Simple database check (use PDO in real application)
        $sql = "SELECT COUNT(*) FROM $table WHERE $column = ?";
        if ($exclude_id) {
            $sql .= " AND id != ?";
        }

        // Execute query and check result
        // If count > 0, value already exists
        $this->errors[] = "$field_name must be unique";

        return $this;
    }
```

- In this example, we support strong passwords.

```
    public function strongPassword($value, $field_name) {
        if (empty($value)) return $this;

        $errors = [];

        if (strlen($value) < 8) {
            $errors[] = "at least 8 characters";
        }
        if (!preg_match('/[A-Z]/', $value)) {
            $errors[] = "at least one uppercase letter";
        }
        if (!preg_match('/[a-z]/', $value)) {
            $errors[] = "at least one lowercase letter";
        }
        if (!preg_match('/[0-9]/', $value)) {
            $errors[] = "at least one number";
        }
        if (!preg_match('/[!@#$%^&*]/', $value)) {
            $errors[] = "at least one special character (!@#$%^&*)";
        }

        if (!empty($errors)) {
            $this->errors[] = "$field_name must have " . implode(', ', $errors);
        }

        return $this;
    }
}
```

Validation Best Practices

1. Always Validate on Server

```
// ❌ Never trust client-side validation alone
<input type="email" required> <!-- Can be bypassed -->

// ✅ Always validate on server
$validator->required($_POST['email'], 'Email')
          ->email($_POST['email'], 'Email');
```

2. Sanitize Input

```php
// Clean input before validation
$input = [
    'name' => trim($_POST['name'] ?? ''),
    'email' => filter_var($_POST['email'] ?? '', FILTER_SANITIZE_EMAIL),
    'age' => filter_var($_POST['age'] ?? '', FILTER_SANITIZE_NUMBER_INT)
];

// Then validate
$validator->required($input['name'], 'Name')
          ->email($input['email'], 'Email')
          ->numeric($input['age'], 'Age');
```

3. Use Type Casting

```php
// After validation, cast to proper types
if (!$validator->hasErrors()) {
    $user_data = [
        'name' => (string)$input['name'],
        'email' => (string)$input['email'],
        'age' => (int)$input['age'],
        'is_active' => (bool)($input['is_active'] ?? false)
    ];
}
```

Frequently Used Validation Patterns

1. Credit Card Numbers

```
public function creditCard($value, $field_name) {
    if (!empty($value)) {
        // Remove spaces and dashes
        $number = preg_replace('/[\s-]/', '', $value);

        // Check if all digits and proper length
        if (!preg_match('/^\d{13,19}$/', $number)) {
            $this->errors[] = "$field_name must be a valid credit card number";
        }
    }
    return $this;
}
```

2. Phone Numbers

```
public function phone($value, $field_name) {
    if (!empty($value)) {
        // Allow various formats: (123) 456-7890, 123-456-7890, 123.456.7890
        $pattern = '/^[\+]?[1-9]?[\d\s\-\(\)\.]{10,15}$/';
        if (!preg_match($pattern, $value)) {
            $this->errors[] = "$field_name must be a valid phone number";
        }
    }
    return $this;
}
```

3. URLs

```
public function url($value, $field_name) {
    if (!empty($value) && !filter_var($value, FILTER_VALIDATE_URL)) {
        $this->errors[] = "$field_name must be a valid URL";
    }
    return $this;
}
```

Key Takeaways

Why Validation is Critical

1. **Security** - Prevents injection attacks and malicious input
2. **Data Integrity** - Ensures consistent, reliable data
3. **User Experience** - Provides clear feedback on errors
4. **Business Logic** - Enforces application rules

Best Practices

- **Server-side validation** is mandatory (client-side is just UX)
- **Method chaining** makes validation code readable
- **Custom rules** can handle specific business requirements
- **Proper error messages** help users fix issues